

15

Authorization: Securing your application

This chapter covers

- Using authorization to control who can use your app
- Using claims-based authorization with policies
- Creating custom policies to handle complex requirements
- Authorizing a request depending upon the resource being accessed
- Hiding elements from a Razor template that the user is unauthorized to access

In chapter 14 I showed you how to add users to an ASP.NET Core application by adding *authentication*. With authentication, users can register and log in to your app using an email address and password. Whenever you add authentication to an app, you inevitably find you want to be able to restrict what some users can do. The process of determining whether a user can perform a given action on your app is called *authorization*.

On an e-commerce site, for example, you may have admin users who are allowed to add new products and change prices, sales users who are allowed to view completed orders, and customer users who are only allowed to place orders and buy products.

In this chapter I'll show you how to use authorization in an app to control what your users can do. In section 15.1 I'll introduce authorization and put it in the context of a real-life scenario you've probably experienced: an airport. I'll describe the sequence of events, from checking in, to passing through security, to entering an airport lounge, and you'll see how these relate to the authorization concepts in this chapter.

In section 15.2 I'll show how authorization fits into an ASP.NET Core web application and how it relates to the `ClaimsPrincipal` class you saw in the previous chapter. You'll see how to enforce the simplest level of authorization in an ASP.NET Core app, ensuring that only authenticated users can execute a Razor Page or MVC action.

We'll extend that approach in section 15.3 by adding in the concept of policies. These let you set specific requirements for a given authenticated user, requiring that they have specific pieces of information in order to execute an action or Razor Page.

You'll use policies extensively in the ASP.NET Core authorization system, so in section 15.4 we'll explore how to handle more complex scenarios. You'll learn about authorization requirements and handlers, and how you can combine them to create specific policies that you can apply to your Razor Pages and actions.

Sometimes, whether a user is authorized depends on which resource or document they're attempting to access. A resource is anything that you're trying to protect, so it could be a document or a post in a social media app. For example, you may allow users to create documents, or to read documents from other users, but only to edit documents that they created themselves. This type of authorization, where you need the details of the document to determine if the user is authorized, is called resource-based authorization, and it's the focus of section 15.5.

In the final section of this chapter, I'll show how you can extend the resource-based authorization approach to your Razor view templates. This lets you modify the UI to hide elements that users aren't authorized to interact with. In particular, you'll see how to hide the Edit button when a user isn't authorized to edit the entity.

We'll start by looking more closely at the concept of authorization, how it differs from authentication, and how it relates to real-life concepts you might see in an airport.

15.1 Introduction to authorization

In this section I provide an introduction to authorization and discuss how it compares to authentication. I use the real-life example of an airport as a case study to illustrate how claims-based authorization works.

For people who are new to web apps and security, authentication and authorization can sometimes be a little daunting. It certainly doesn't help that the words look so similar! The two concepts are often used together, but they're definitely distinct:

- *Authentication*—The process of determining who made a request
- *Authorization*—The process of determining whether the requested action is allowed

Typically, authentication occurs first, so that you know who is making a request to your app. For traditional web apps, your app authenticates a request by checking the encrypted cookie that was set when the user logged in (as you saw in the previous chapter). Web APIs typically use a header instead of a cookie for authentication, but the process is the same.

Once a request is authenticated and you know who is making the request, you can determine whether they're allowed to execute an action on your server. This process is called authorization and is the focus of this chapter.

Before we dive into code and start looking at authorization in ASP.NET Core, I'll put these concepts into a real-life scenario you're hopefully familiar with: checking in at an airport. To enter an airport and board a plane, you must pass through several steps: an initial step to prove who you are (authentication); and subsequent steps that check whether you're allowed to proceed (authorization). In simplified form, these might look like this:

- 1 Show your passport at the check-in desk. Receive a boarding pass.
- 2 Show your boarding pass to enter security. Pass through security.
- 3 Show your frequent flyer card to enter the airline lounge. Enter the lounge.
- 4 Show your boarding pass to board the flight. Enter the airplane.

Obviously, these steps, also shown in figure 15.1, will vary somewhat in real life (I don't have a frequent flyer card!), but we'll go with them for now. Let's explore each step a little further.

When you arrive at the airport, the first thing you do is go to the check-in counter. Here, you can purchase a plane ticket, but to do so, you need to prove who you are by providing a passport; you *authenticate* yourself. If you've forgotten your passport, you can't authenticate, and you can't go any further.

Once you've purchased your ticket, you're issued a boarding pass, which says which flight you're on. We'll assume it also includes a `BoardingPassNumber`. You can think of this number as an additional *claim* associated with your identity.

DEFINITION A *claim* is a piece of information about a user that consists of a *type* and an optional *value*.

The next step is security. The security guards will ask you to present your boarding pass for inspection, which they'll use to check that you have a flight and so are allowed deeper into the airport. This is an authorization process: you must have the required claim (a `BoardingPassNumber`) to proceed.

If you don't have a valid `BoardingPassNumber`, there are two possibilities for what happens next:

- *If you haven't yet purchased a ticket*—You'll be directed back to the check-in desk, where you can authenticate and purchase a ticket. At that point, you can try to enter security again.

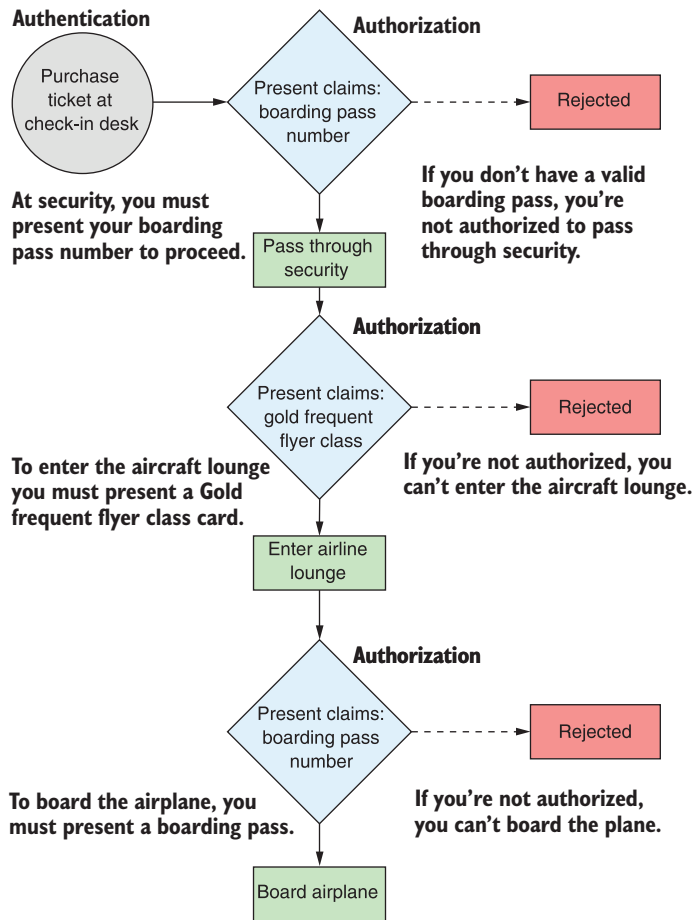


Figure 15.1 When boarding a plane at an airport, you pass through several authorization steps. At each authorization step, you must present a claim in the form of a boarding pass or a frequent flyer card. If you're not authorized, access will be denied.

- *If you have an invalid ticket*—You won't be allowed through security, and there's nothing else you can do. If, for example, you show up with a boarding pass a week late for your flight, they probably won't let you through. (Ask me how I know!)

Once you're through security, you need to wait for your flight to start boarding, but unfortunately there aren't any seats free. Typical! Luckily, you're a regular flyer, and you've notched up enough miles to achieve a Gold frequent flyer status, so you can use the airline lounge.

You head to the lounge, where you're asked to present your Gold Frequent Flyer card to the attendant, and they let you in. This is another example of authorization. You must have a `FrequentFlyerClass` claim with a value of `Gold` to proceed.

NOTE You've used authorization twice so far in this scenario. Each time, you presented a claim to proceed. In the first case, the presence of any `BoardingPassNumber` was sufficient, whereas for the `FrequentFlyerClass` claim, you needed the specific value of `Gold`.

When you're boarding the airplane, you have one final authorization step, in which you must present the `BoardingPassNumber` claim again. You presented this claim earlier, but boarding the aircraft is a distinct action from entering security, so you have to present it again.

This whole scenario has lots of parallels with requests to a web app:

- Both processes start with authentication.
- You have to prove *who* you are in order to retrieve the *claims* you need for authorization.
- You use authorization to protect sensitive actions like entering security and the airline lounge.

I'll reuse this airport scenario throughout the chapter to build a simple web application that simulates the steps you take in an airport. We've covered the concept of authorization in general, so in the next section we'll look at how authorization works in ASP.NET Core. We'll start with the most basic level of authorization, ensuring only authenticated users can execute an action, and look at what happens when you try to execute such an action.

15.2 Authorization in ASP.NET Core

In this section you'll see how the authorization principles described in the previous section apply to an ASP.NET Core application. You'll learn about the role of the `[Authorize]` attribute and `AuthorizationMiddleware` in authorizing requests to Razor Pages and MVC actions. Finally, you'll learn about the process of preventing unauthenticated users from executing endpoints, and what happens when users are unauthorized.

The ASP.NET Core framework has authorization built in, so you can use it anywhere in your app, but it's most common in ASP.NET Core 5.0 to apply authorization via the `AuthorizationMiddleware`. The `AuthorizationMiddleware` should be placed *after* both the routing middleware and the authentication middleware, but *before* the endpoint middleware, as shown in figure 15.2.

NOTE In ASP.NET Core, an *endpoint* refers to the handler selected by the routing middleware, which will generate a response when executed. It is typically a Razor Page or a Web API action method.

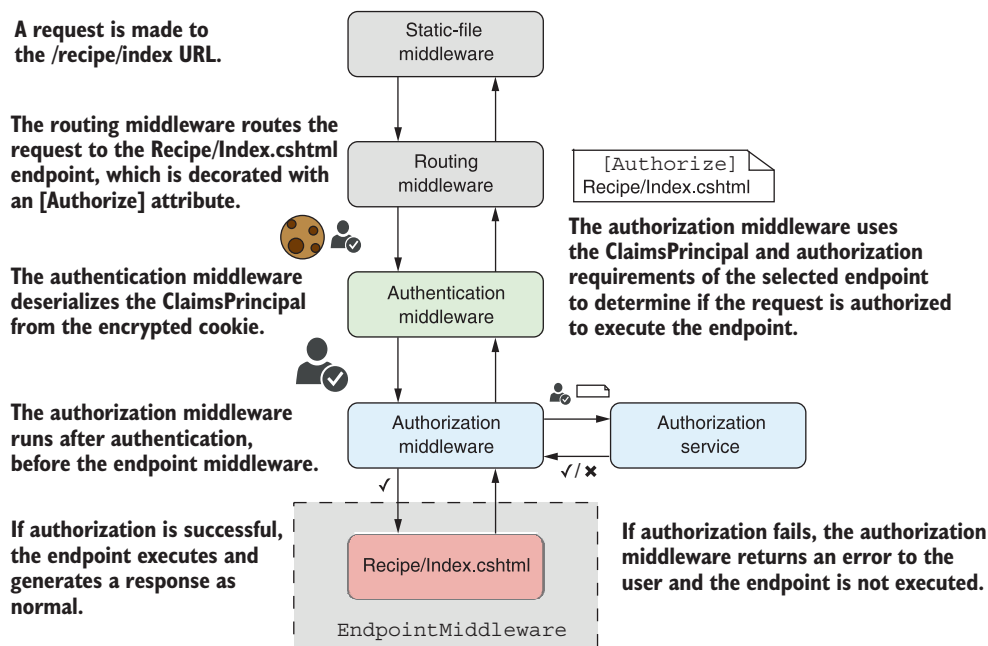


Figure 15.2 Authorization occurs after an endpoint has been selected and after the request is authenticated, but before the action method or Razor Page endpoint is executed.

With this configuration, the `RoutingMiddleware` selects an endpoint to execute based on the request's URL, such as a Razor Page, as you saw in chapter 5. Metadata about the selected endpoint is available to all middleware that occurs after the routing middleware. This metadata includes details about any authorization requirements for the endpoint, and it's typically attached by decorating an action or Razor Page with an `[Authorize]` attribute.

The `AuthenticationMiddleware` deserializes the encrypted cookie (or bearer token for APIs) associated with the request to create a `ClaimsPrincipal`. This object is set as the `HttpContext.User` for the request, so all subsequent middleware can access this value. It contains all the Claims that were added to the cookie when the user authenticated.

Now we come to the `AuthorizationMiddleware`. This middleware checks if the selected endpoint has any authorization requirements, based on the metadata provided by the `RoutingMiddleware`. If the endpoint has authorization requirements, the `AuthorizationMiddleware` uses the `HttpContext.User` to determine if the current request is authorized to execute the endpoint.

If the request is authorized, the next middleware in the pipeline executes as normal. If the request is not authorized, the `AuthorizationMiddleware` short-circuits the middleware pipeline, and the endpoint middleware is never executed.

NOTE The order of middleware in your pipeline is very important. The call to `UseAuthorization()` must come after `UseRouting()` and `UseAuthentication()`, but before `UseEndpoints()`.

Changes to authorization in ASP.NET Core 3.0

The authorization system changed significantly in ASP.NET Core 3.0. Prior to this release, the `AuthorizationMiddleware` did not exist. Instead, the `[Authorize]` attribute executed the authorization logic as part of the MVC filter pipeline.

In practice, from the point of view of using authorization in your actions and Razor Pages, there is no real difference from a developer's point of view. Why change it then?

The new design, using the `AuthorizationMiddleware` in conjunction with endpoint routing (introduced at the same time), enables additional scenarios. The changes make it easier to apply authorization to non-MVC/Razor Page endpoints. You'll see how to create these types of endpoints in chapter 19. You can also read more about the authorization changes in the "Authorization" section of Microsoft's "Migrate from ASP.NET Core 2.2 to 3.0" documentation: <http://mng.bz/1rvj>.

The `AuthorizationMiddleware` is responsible for applying authorization requirements and ensuring that only authorized users can execute protected endpoints. In section 15.2.1 you'll learn how to apply the simplest authorization requirement, and in section 15.2.2 you'll see how the framework responds when a user is not authorized to execute an endpoint.

15.2.1 Preventing anonymous users from accessing your application

When you think about authorization, you typically think about checking whether a particular user has permission to execute an endpoint. In ASP.NET Core you normally achieve this by checking whether a user has a given claim.

There's an even more basic level of authorization we haven't considered yet—only allowing authenticated users to execute an endpoint. This is even simpler than the claims scenario (which we'll come to later) as there are only two possibilities:

- *The user is authenticated*—The action executes as normal.
- *The user is unauthenticated*—The user can't execute the endpoint.

You can achieve this basic level of authorization by using the `[Authorize]` attribute, which you saw in chapter 13 when we discussed authorization filters. You can apply this attribute to your actions and Razor Pages, as shown in the following listing, to restrict them to authenticated (logged-in) users only. If an unauthenticated user tries to execute an action or Razor Page protected with the `[Authorize]` attribute, they'll be redirected to the login page.

Listing 15.1 Applying [Authorize] to an action

```
public class RecipeApiController : ControllerBase
{
    public IActionResult List()
    {
        return Ok();
    }

    [Authorize]
    public IActionResult View()
    {
        return Ok();
    }
}
```

This action can be executed by anyone, even when not logged in.

Applies [Authorize] to individual actions, whole controllers, or Razor Pages

This action can only be executed by authenticated users.

Applying the [Authorize] attribute to an endpoint attaches metadata to it, indicating only authenticated users may access the endpoint. As you saw in figure 15.2, this metadata is made available to the AuthorizationMiddleware when an endpoint is selected by the RoutingMiddleware.

You can apply the [Authorize] attribute at the action scope, controller scope, Razor Page scope, or globally, as you saw in chapter 13. Any action or Razor Page that has the [Authorize] attribute applied in this way can be executed only by an authenticated user. Unauthenticated users will be redirected to the login page.

TIP There are several different ways to apply the [Authorize] attribute globally. You can read about the different options, and when to choose which option, on my blog: <http://mng.bz/opQp>.

Sometimes, especially when you apply the [Authorize] attribute globally, you might need to poke holes in this authorization requirement. If you apply the [Authorize] attribute globally, then any unauthenticated request will be redirected to the login page for your app. But if the [Authorize] attribute is global, then when the login page tries to load, you'll be unauthenticated and redirected to the login page again. And now you're stuck in an infinite redirect loop.

To get around this, you can designate specific endpoints to ignore the [Authorize] attribute by applying the [AllowAnonymous] attribute to an action or Razor Page, as shown next. This allows unauthenticated users to execute the action, so you can avoid the redirect loop that would otherwise result.

Listing 15.2 Applying [AllowAnonymous] to allow unauthenticated access

```
[Authorize]
public class AccountController : ControllerBase
{
    public IActionResult ManageAccount()
    {
        return Ok();
    }
}
```

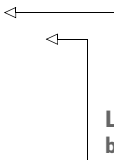
Applied at the controller scope, so the user must be authenticated for all actions on the controller.

Only authenticated users may execute ManageAccount.


```

[AllowAnonymous]
public IActionResult Login()
{
    return Ok();
}

```



[AllowAnonymous]
overrides **[Authorize]** to
allow unauthenticated users.

**Login can be executed
by anonymous users.**

WARNING If you apply the `[Authorize]` attribute globally, be sure to add the `[AllowAnonymous]` attribute to your login actions, error actions, password reset actions, and any other actions that you need unauthenticated users to execute. If you're using the default Identity UI described in chapter 14, this is already configured for you.

If an unauthenticated user attempts to execute an action protected by the `[Authorize]` attribute, traditional web apps will redirect them to the login page. But what about Web APIs? And what about more complex scenarios, where a user is logged in but doesn't have the necessary claims to execute an action? In section 15.2.2 we'll look at how the ASP.NET Core authentication services handle all of this for you.

15.2.2 Handling unauthorized requests

In the previous section you saw how to apply the `[Authorize]` attribute to an action to ensure only authenticated users can execute it. In section 15.3 we'll look at more complex examples that require you to also have a specific claim. In both cases, you must meet one or more authorization requirements (for example, you must be authenticated) to execute the action.

If the user meets the authorization requirements, then the request passes unimpeded through the `AuthorizationMiddleware`, and the endpoint is executed in the `EndpointMiddleware`. If they don't meet the requirements for the selected endpoint, the `AuthorizationMiddleware` will short-circuit the request. Depending on why the request failed authorization, the `AuthorizationMiddleware` generates one of two different types of responses, as shown in figure 15.3:

- *Challenge*—This response indicates the user was not authorized to execute the action because they weren't yet logged in.
- *Forbid*—This response indicates that the user was logged in but didn't meet the requirements to execute the action. They didn't have a required claim, for example.

NOTE If you apply the `[Authorize]` attribute in basic form, as you did in section 15.2.1, you will only generate challenge responses. In this case, a challenge response will be generated for unauthenticated users, but authenticated users will always be authorized.

The exact HTTP response generated by a challenge or forbid response typically depends on the type of application you're building and so the type of authentication your application uses: a traditional web application with Razor Pages, or an API application.

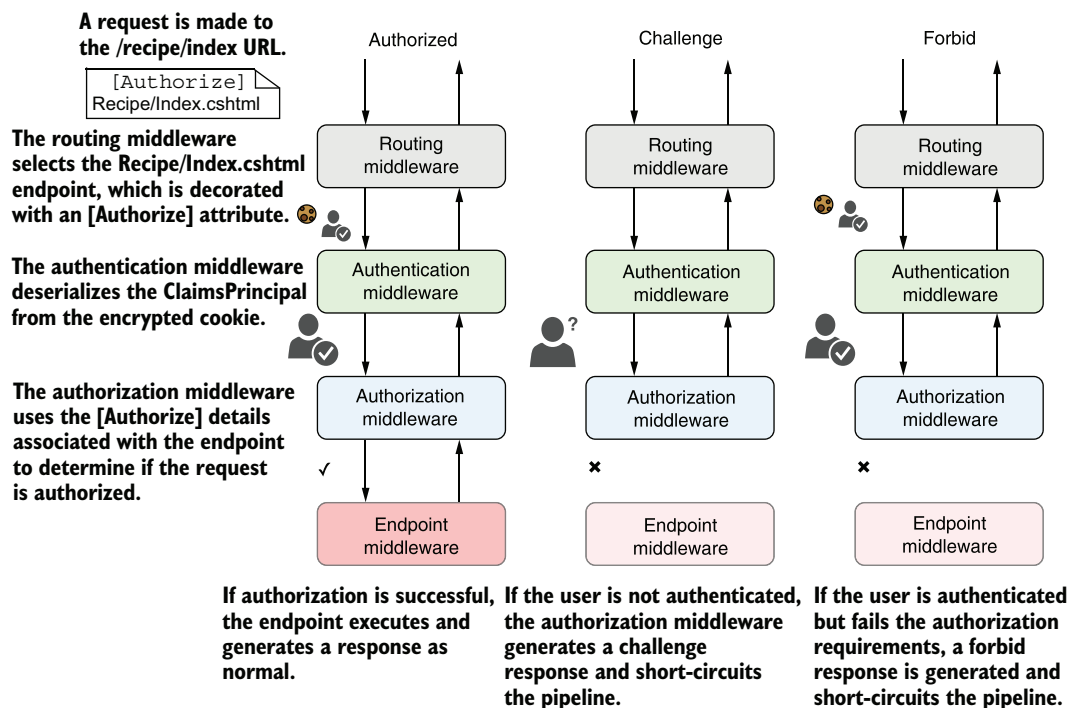


Figure 15.3 The three types of response to an authorization attempt. In the left example, the request contains an authentication cookie, so the user is authenticated in the `AuthenticationMiddleware`. The `AuthorizationMiddleware` confirms the authenticated user can access the selected endpoint, so the endpoint is executed. In the center example, the request is not authenticated, so the `AuthorizationMiddleware` generates a challenge response and short-circuits the pipeline. In the right example, the request is authenticated, but the user does not have permission to execute the endpoint, so a forbid response is generated.

For traditional web apps using cookie authentication, such as when you use ASP.NET Core Identity, as in chapter 14, the challenge and forbid responses generate an HTTP redirect to a page in your application.

A challenge response indicates the user isn't yet authenticated, so they're redirected to the login page for the app. After logging in, they can attempt to execute the protected resource again.

A forbid response means the request was from a user that *already* logged in, but they're still not allowed to execute the action. Consequently, the user is redirected to a "forbidden" or "access denied" web page, as shown in figure 15.4, which informs them they can't execute the action or Razor Page.

The preceding behavior is standard for traditional web apps, but Web APIs typically use a different approach to authentication, as you saw in chapter 14. Instead of logging in and using the API directly, you'd typically log in to a third-party application

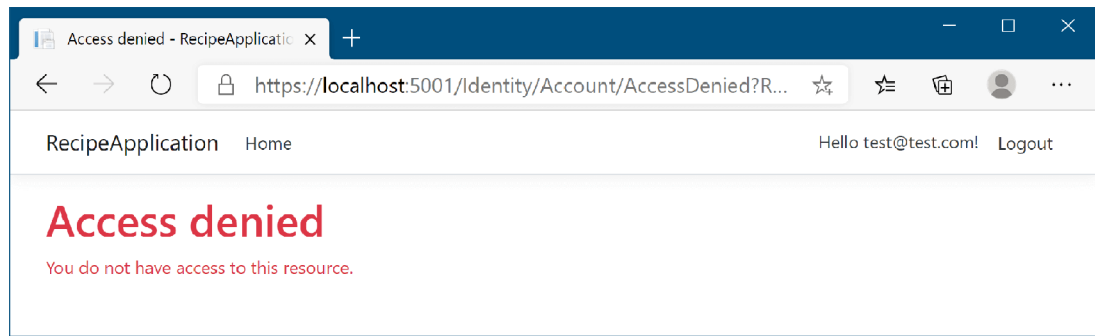


Figure 15.4 A forbid response in traditional web apps using cookie authentication. If you don't have permission to execute a Razor Page and you're already logged in, you'll be redirected to an "access denied" page.

that provides a token to the client-side SPA or mobile app. The client-side app sends this token when it makes a request to your Web API.

Authenticating a request for a Web API using tokens is essentially identical to a traditional web app that uses cookies; `AuthenticationMiddleware` deserializes the cookie or token to create the `ClaimsPrincipal`. The difference is in how a Web API handles authorization failures.

When a Web API app generates a challenge response, it returns a 401 Unauthorized error response to the caller. Similarly, when the app generates a forbid response, it returns a 403 Forbidden response. The traditional web app essentially handled these errors by automatically redirecting unauthorized users to the login or "access denied" page, but the Web API doesn't do this. It's up to the client-side SPA or mobile app to detect these errors and handle them as appropriate.

TIP The difference in authorization behavior is one of the reasons I generally recommend creating separate apps for your APIs and Razor pages apps—it's possible to have both in the same app, but the configuration is more complex.

The different behavior between traditional web apps and SPAs can be confusing initially, but you generally don't need to worry about that too much in practice. Whether you're building a Web API or a traditional MVC web app, the authorization code in your app looks the same in both cases. Apply `[Authorize]` attributes to your endpoints, and let the framework take care of the differences for you.

NOTE In chapter 14 you saw how to configure ASP.NET Core Identity in a Razor Pages app. This chapter assumes you're building a Razor Pages app too, but the chapter is equally applicable if you're building a Web API. Authorization policies are applied in the same way, whichever style of app you're building. It's only the final response of unauthorized requests that differs.

You've seen how to apply the most basic authorization requirement—restricting an endpoint to authenticated users only—but most apps need something more subtle than this all-or-nothing approach.

Consider the airport scenario from section 15.1. Being authenticated (having a passport) isn't enough to get you through security. Instead, you also need a specific claim: `BoardingPassNumber`. In the next section we'll look at how you can implement a similar requirement in ASP.NET Core.

15.3 Using policies for claims-based authorization

In the previous section, you saw how to require that users be logged in to access an endpoint. In this section you'll see how to apply additional requirements. You'll learn to use authorization policies to perform claims-based authorization to require that a logged in user have the required claims to execute a given endpoint.

In chapter 14 you saw that authentication in ASP.NET Core centers around a `ClaimsPrincipal` object, which represents the user. This object has a collection of claims that contain pieces of information about the user, such as their name, email, and date of birth.

You can use these to customize the app for each user, by displaying a welcome message addressing the user by name, for example, but you can also use claims for authorization. For example, you might only authorize a user if they have a specific claim (such as `BoardingPassNumber`) or if a claim has a specific value (`FrequentFlyerClass` claim with the value `Gold`).

In ASP.NET Core the rules that define whether a user is authorized are encapsulated in a *policy*.

DEFINITION A *policy* defines the requirements you must meet for a request to be authorized.

Policies can be applied to an action using the `[Authorize]` attribute, similar to the way you saw in section 15.2.1. This listing shows a Razor Page `PageModel` that represents the first authorization step in the airport scenario. The `AirportSecurity.cshtml` Razor Page is protected by an `[Authorize]` attribute, but you've also provided a policy name: `"CanEnterSecurity"`.

Listing 15.3 Applying an authorization policy to a Razor Page

```
[Authorize("CanEnterSecurity")]
public class AirportSecurityModel : PageModel
{
    public void OnGet()
    {
    }
}
```

← Applying the "CanEnterSecurity" policy using `[Authorize]`

← Only users that satisfy the "CanEnterSecurity" policy can execute the Razor Page.

If a user attempts to execute the `AirportSecurity.cshtml` Razor Page, the authorization middleware will verify whether the user satisfies the policy's requirements (we'll look at the policy itself shortly). This gives one of three possible outcomes:

- *The user satisfies the policy*—The middleware pipeline continues, and the `Endpoint-Middleware` executes the Razor Page as normal.
- *The user is unauthenticated*—The user is redirected to the login page.
- *The user is authenticated but doesn't satisfy the policy*—The user is redirected to a “forbidden” or “access denied” page.

These three outcomes correlate with the real-life outcomes you might expect when trying to pass through security at the airport:

- *You have a valid boarding pass*—You can enter security as normal.
- *You don't have a boarding pass*—You're redirected to purchase a ticket.
- *Your boarding pass is invalid (you turned up a day late, for example)*—You're blocked from entering.

Listing 15.3 shows how you can apply a policy to a Razor Page using the `[Authorize]` attribute, but you still need to define the `CanEnterSecurity` policy.

You add policies to an ASP.NET Core application in the `ConfigureServices` method of `Startup.cs`, as shown in listing 15.4. First you add the authorization services using `AddAuthorization()`, and then you can add policies by calling `AddPolicy()` on the `AuthorizationOptions` object. You define the policy itself by calling methods on a provided `AuthorizationPolicyBuilder` (called `policyBuilder` here).

Listing 15.4 Adding an authorization policy using `AuthorizationPolicyBuilder`

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthorization(options =>           ← Calls AddAuthorization to
    {                                               configure AuthorizationOptions
        options.AddPolicy(
            "CanEnterSecurity",                    ← Provides a name for the policy
            policyBuilder => policyBuilder
                .RequireClaim("BoardingPassNumber")); ← Defines the policy
    });                                           requirements using
    // Additional service configuration           AuthorizationPolicyBuilder
}
```

Adds a new policy →

When you call `AddPolicy` you provide a name for the policy, which should match the value you use in your `[Authorize]` attributes, and you define the requirements of the policy. In this example, you have a single simple requirement: the user must have a claim of type `BoardingPassNumber`. If a user has this claim, whatever its value, the policy will be satisfied and the user will be authorized.

REMEMBER A *claim* is information about the user, as a key-value pair. A policy defines the requirements for successful authorization. A policy can require

that a user have a given claim, as well as specify more complex requirements, as you'll see shortly.

`AuthorizationPolicyBuilder` contains several methods for creating simple policies like this, as shown in table 15.1. For example, an overload of the `RequireClaim()` method lets you specify a specific value that a claim must have. The following would let you create a policy where the "BoardingPassNumber" claim must have a value of "A1234":

```
policyBuilder => policyBuilder.RequireClaim("BoardingPassNumber", "A1234");
```

Table 15.1 Simple policy builder methods on `AuthorizationPolicyBuilder`

Method	Policy behavior
<code>RequireAuthenticatedUser()</code>	The required user must be authenticated. Creates a policy similar to the default <code>[Authorize]</code> attribute, where you don't set a policy.
<code>RequireClaim(claim, values)</code>	The user must have the specified claim. If provided, the claim must be one of the specified values.
<code>RequireUsername(username)</code>	The user must have the specified username.
<code>RequireAssertion(function)</code>	Executes the provided lambda function, which returns a <code>bool</code> , indicating whether the policy was satisfied.

Role-based authorization vs. claims-based authorization

If you look at all of the methods available on the `AuthorizationPolicyBuilder` type using `IntelliSense`, you might notice that there's a method I didn't mention in table 15.1, `RequireRole()`. This is a remnant of the role-based approach to authorization used in previous versions of ASP.NET, and I don't recommend using it.

Before Microsoft adopted the claims-based authorization used by ASP.NET Core and recent versions of ASP.NET, role-based authorization was the norm. Users were assigned to one or more roles, such as `Administrator` or `Manager`, and authorization involved checking whether the current user was in the required role.

This role-based approach to authorization is possible in ASP.NET Core, but it's primarily used for legacy compatibility reasons. Claims-based authorization is the suggested approach. Unless you're porting a legacy app that uses roles, I suggest you embrace claims-based authorization and leave those roles behind.

You can use these methods to build simple policies that can handle basic situations, but often you'll need something more complicated. What if you wanted to create a policy that enforces that only users over the age of 18 can execute an endpoint?

The `DateOfBirth` claim provides the information you need, but there's not a *single* correct value, so you couldn't use the `RequireClaim()` method. You *could* use the

`RequireAssertion()` method and provide a function that calculates the age from the `DateOfBirth` claim, but that could get messy pretty quickly.

For more complex policies that can't be easily defined using the `RequireClaim()` method, I recommend you take a different approach and create a custom policy, as you'll see in the following section.

15.4 Creating custom policies for authorization

You've already seen how to create a policy by requiring a specific claim, or requiring a specific claim with a specific value, but often the requirements will be more complex than that. In this section you'll learn how to create custom authorization requirements and handlers. You'll also see how to configure authorization requirements where there are multiple ways to satisfy a policy, any of which are valid.

Let's return to the airport example. You've already configured the policy for passing through security, and now you're going to configure the policy that controls whether you're authorized to enter the airline lounge.

As you saw in figure 15.1, you're allowed to enter the lounge if you have a `FrequentFlyerClass` claim with a value of `Gold`. If this was the only requirement, you could use `AuthorizationPolicyBuilder` to create a policy like this:

```
options.AddPolicy("CanAccessLounge", policyBuilder =>
    policyBuilder.RequireClaim("FrequentFlyerClass", "Gold");
```

But what if the requirements are more complicated than this? For example, suppose you can enter the lounge if you're at least 18 years old (as calculated from the `DateOfBirth` claim) and you're one of the following:

- You're a gold-class frequent flyer (have a `FrequentFlyerClass` claim with value `"Gold"`)
- You're an employee of the airline (have an `EmployeeNumber` claim)

If you've ever been banned from the lounge (you have an `IsBannedFromLounge` claim), you won't be allowed in, even if you satisfy the other requirements.

There's no way of achieving this complex set of requirements with the basic usage of `AuthorizationPolicyBuilder` you've seen so far. Luckily, these methods are a wrapper around a set of building blocks that you can combine to achieve the desired policy.

15.4.1 Requirements and handlers: The building blocks of a policy

Every policy in ASP.NET Core consists of one or more *requirements*, and every requirement can have one or more *handlers*. For the airport lounge example, you have a single policy (`"CanAccessLounge"`), two requirements (`MinimumAgeRequirement` and `AllowedInLoungeRequirement`), and several handlers, as shown in figure 15.5.

For a policy to be satisfied, a user must fulfill *all* the requirements. If the user fails *any* of the requirements, the authorize middleware won't allow the protected endpoint

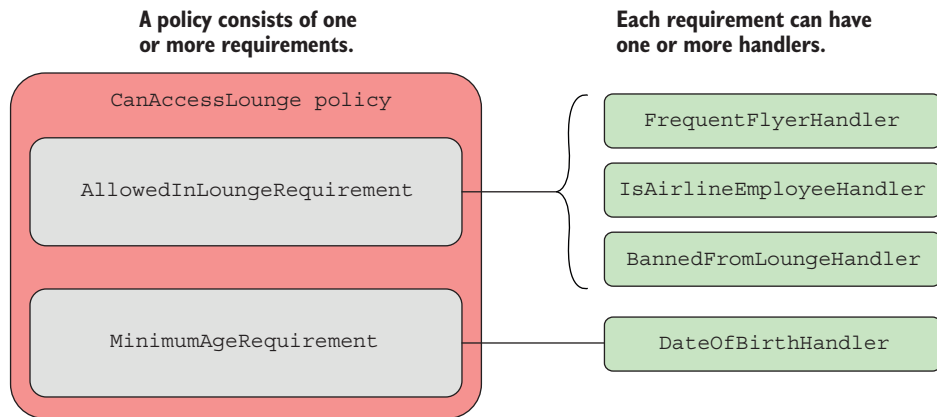


Figure 15.5 A policy can have many requirements, and every requirement can have many handlers. By combining multiple requirements in a policy, and by providing multiple handler implementations, you can create complex authorization policies that meet any of your business requirements.

to be executed. In this example, a user must be allowed to access the lounge *and* must be over 18 years old.

Each requirement can have one or more handlers, which will confirm that the requirement has been satisfied. For example, as shown in figure 15.5, `AllowedInLoungeRequirement` has two handlers that can satisfy the requirement:

- `FrequentFlyerHandler`
- `IsAirlineEmployeeHandler`

If the user satisfies either of these handlers, then `AllowedInLoungeRequirement` is satisfied. You don't need all handlers for a requirement to be satisfied, you just need one.

NOTE Figure 15.5 shows a third handler, `BannedFromLoungeHandler`, which I'll cover in section 15.4.2. It's slightly different in that it can only *fail* a requirement, not *satisfy* it.

You can use requirements and handlers to achieve most any combination of behavior you need for a policy. By combining handlers for a requirement, you can validate conditions using a logical OR: if any of the handlers are satisfied, the requirement is satisfied. By combining requirements, you create a logical AND: all the requirements must be satisfied for the policy to be satisfied, as shown in figure 15.6.

TIP You can also add multiple policies to a Razor Page or action method by applying the `[Authorize]` attribute multiple times; for example, `[Authorize("Policy1"), Authorize("Policy2")]`. All policies must be satisfied for the request to be authorized.

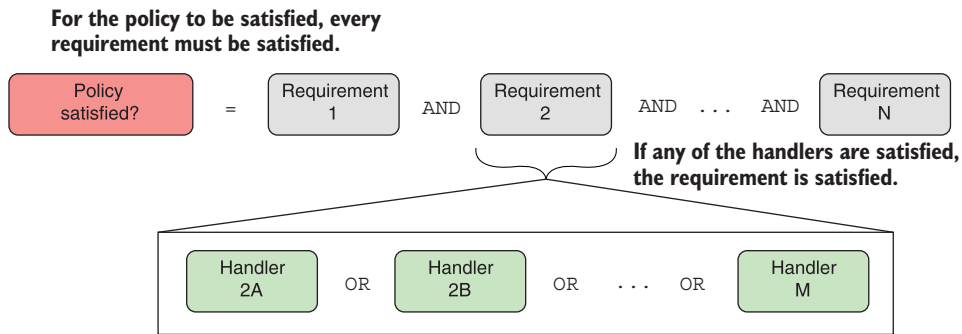


Figure 15.6 For a policy to be satisfied, every requirement must be satisfied. A requirement is satisfied if any of the handlers are satisfied.

I've highlighted requirements and handlers that will make up your "CanAccessLounge" policy, so in the next section you'll build each of the components and apply them to the airport sample app.

15.4.2 Creating a policy with a custom requirement and handler

You've seen all the pieces that make up a custom authorization policy, so in this section we'll explore the implementation of the "CanAccessLounge" policy.

CREATING AN `IAuthorizationRequirement` TO REPRESENT A REQUIREMENT

As you've seen, a custom policy can have multiple requirements, but what *is* a requirement in code terms? Authorization requirements in ASP.NET Core are any class that implements the `IAuthorizationRequirement` interface. This is a blank, marker interface, which you can apply to any class to indicate that it represents a requirement.

If the interface doesn't have any members, you might be wondering what the requirement class needs to look like. Typically, they're simple POCO classes. The following listing shows `AllowedInLoungeRequirement`, which is about as simple as a requirement can get. It has no properties or methods; it implements the required `IAuthorizationRequirement` interface.

Listing 15.5 `AllowedInLoungeRequirement`

```
public class AllowedInLoungeRequirement
    : IAuthorizationRequirement { }
```

The interface identifies the class as an authorization requirement.

This is the simplest form of requirement, but it's also common for them to have one or two properties that make the requirement more generalized. For example, instead of creating the highly specific `MustBe18YearsOldRequirement`, you could instead create a parameterized `MinimumAgeRequirement`, as shown in the following listing. By providing the minimum age as a parameter to the requirement, you can reuse the requirement for other policies with different minimum age requirements.

Listing 15.6 The parameterized MinimumAgeRequirement

```
public class MinimumAgeRequirement : IAuthorizationRequirement
{
    public MinimumAgeRequirement(int minimumAge)
    {
        MinimumAge = minimumAge;
    }
    public int MinimumAge { get; }
}
```

The interface identifies the class as an authorization requirement.

The minimum age is provided when the requirement is created.

Handlers can use the exposed minimum age to determine whether the requirement is satisfied.

The requirements are the easy part. They represent each of the components of the policy that must be satisfied for the policy to be satisfied overall.

CREATING A POLICY WITH MULTIPLE REQUIREMENTS

You've created the two requirements, so now you can configure the "CanAccessLounge" policy to use them. You configure your policies as you did before, in the `ConfigureServices` method of `Startup.cs`. Listing 15.7 shows how to do this by creating an instance of each requirement and passing them to `AuthorizationPolicyBuilder`. The authorization handlers will use these requirement objects when attempting to authorize the policy.

Listing 15.7 Creating an authorization policy with multiple requirements

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthorization(options =>
    {
        options.AddPolicy(
            "CanEnterSecurity",
            policyBuilder => policyBuilder
                .RequireClaim(Claims.BoardingPassNumber));
        options.AddPolicy(
            "CanAccessLounge",
            policyBuilder => policyBuilder.AddRequirements(
                new MinimumAgeRequirement(18),
                new AllowedInLoungeRequirement()
            ));
    });
    // Additional service configuration
}
```

Adds a new policy for the airport lounge, called `CanAccessLounge`

Adds the previous simple policy for passing through security

Adds an instance of each `IAuthorizationRequirement` object

You now have a policy called "CanAccessLounge" with two requirements, so you can apply it to a Razor Page or action method using the `[Authorize]` attribute, in exactly the same way you did for the "CanEnterSecurity" policy:

```
[Authorize("CanAccessLounge")]
public class AirportLoungeModel : PageModel
{
}
```

```

    public void OnGet() { }
}

```

When a request is routed to the `AirportLounge.cshtml` Razor Page, the authorize middleware executes the authorization policy and each of the requirements is inspected. But you saw earlier that the requirements are purely data; they indicate what needs to be fulfilled, but they don't describe how that has to happen. For that, you need to write some handlers.

CREATING AUTHORIZATION HANDLERS TO SATISFY YOUR REQUIREMENTS

Authorization handlers contain the logic of how a specific `IAuthorizationRequirement` can be satisfied. When executed, a handler can do one of three things:

- Mark the requirement handling as a success
- Not do anything
- Explicitly fail the requirement

Handlers should implement `AuthorizationHandler<T>`, where `T` is the type of requirement they handle. For example, the following listing shows a handler for `AllowedInLoungeRequirement` that checks whether the user has a claim called `FrequentFlyerClass` with a value of `Gold`.

Listing 15.8 `FrequentFlyerHandler` for `AllowedInLoungeRequirement`

You must override the abstract `HandleRequirementAsync` method.

```

public class FrequentFlyerHandler :
    AuthorizationHandler<AllowedInLoungeRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        AllowedInLoungeRequirement requirement)
    {
        if (context.User.HasClaim("FrequentFlyerClass", "Gold"))
        {
            context.Succeed(requirement);
        }
        return Task.CompletedTask;
    }
}

```

The context contains details such as the `ClaimsPrincipal` user object.

The handler implements `AuthorizationHandler<T>`.

The requirement instance to handle

Checks whether the user has the `FrequentFlyerClass` claim with the `Gold` value

If the user had the necessary claim, then mark the requirement as satisfied by calling `Succeed`.

If the requirement wasn't satisfied, do nothing.

This handler is functionally equivalent to the simple `RequireClaim()` handler you saw at the start of section 15.4, but using the requirement and handler approach instead.

When a request is routed to the `AirportLounge.cshtml` Razor Page, the authorization middleware sees the `[Authorize]` attribute on the endpoint with the "CanAccess-Lounge" policy. It loops through all the requirements in the policy, and all the handlers for each requirement, calling the `HandleRequirementAsync` method for each.

The authorization middleware passes the current `AuthorizationHandlerContext` and the requirement to be checked to each handler. The current `ClaimsPrincipal`

being authorized is exposed on the context as the `User` property. In listing 15.8, `FrequentFlyerHandler` uses the context to check for a claim called `FrequentFlyerClass` with the `Gold` value, and if it exists, indicates that the user is allowed to enter the airline lounge by calling `Succeed()`.

NOTE Handlers mark a requirement as being successfully satisfied by calling `context.Succeed()` and passing the requirement as an argument.

It's important to note the behavior when the user *doesn't* have the claim. `FrequentFlyerHandler` doesn't do anything if this is the case (it returns a completed `Task` to satisfy the method signature).

NOTE Remember, if any of the handlers associated with a requirement pass, then the requirement is a success. Only *one* of the handlers must succeed for the requirement to be satisfied.

This behavior, whereby you either call `context.Succeed()` or do nothing, is typical for authorization handlers. The following listing shows the implementation of `IsAirlineEmployeeHandler`, which uses a similar claim check to determine whether the requirement is satisfied.

Listing 15.9 `IsAirlineEmployeeHandler`

```
public class IsAirlineEmployeeHandler :
    AuthorizationHandler<AllowedInLoungeRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        AllowedInLoungeRequirement requirement)
    {
        if(context.User.HasClaim(c => c.Type == "EmployeeNumber"))
        {
            context.Succeed(requirement);
        }
        return Task.CompletedTask;
    }
}
```

The handler implements `AuthorizationHandler<T>`.

You must override the abstract `HandleRequirementAsync` method.

If the user has the necessary claim, mark the requirement as satisfied by calling `Succeed`.

Checks whether the user has the `EmployeeNumber` claim

If the requirement wasn't satisfied, do nothing.

TIP It's possible to write very generic handlers that can be used with multiple requirements, but I suggest sticking to handling a single requirement only. If you need to extract some common functionality, move it to an external service and call that from both handlers.

This pattern of authorization handler is common,¹ but in some cases, instead of checking for a *success* condition, you might want to check for a *failure* condition. In the

¹ I'll leave the implementation of `MinimumAgeHandler` for `MinimumAgeRequirement` as an exercise. You can find an example in the code samples for the chapter.

airport example, you don't want to authorize someone who was previously banned from the lounge, even if they would otherwise be allowed to enter.

You can handle this scenario by using the `context.Fail()` method exposed on the context, as shown in the following listing. Calling `Fail()` in a handler will always cause the requirement, and hence the whole policy, to fail. You should only use it when you want to guarantee failure, even if other handlers indicate success.

Listing 15.10 Calling `context.Fail()` in a handler to fail the requirement

```
public class BannedFromLoungeHandler :
    AuthorizationHandler<AllowedInLoungeRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        AllowedInLoungeRequirement requirement)
    {
        if (context.User.HasClaim(c => c.Type == "IsBanned"))
        {
            context.Fail();
        }

        return Task.CompletedTask;
    }
}
```

The handler implements `AuthorizationHandler<T>`.

You must override the abstract `HandleRequirementAsync` method.

Checks whether the user has the `IsBanned` claim

If the user has the claim, fail the requirement by calling `Fail`. The whole policy will fail.

If the claim wasn't found, do nothing.

In most cases, your handlers will either call `Succeed()` or will do nothing, but the `Fail()` method is useful when you need a kill-switch to guarantee that a requirement won't be satisfied.

NOTE Whether a handler calls `Succeed()`, `Fail()`, or neither, the authorization system will always execute all of the handlers for a requirement, and all the requirements for a policy, so you can be sure your handlers will always be called.

The final step to complete your authorization implementation for the app is to register the authorization handlers with the DI container, as shown in the following listing.

Listing 15.11 Registering the authorization handlers with the DI container

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthorization(options =>
    {
        options.AddPolicy(
            "CanEnterSecurity",
            policyBuilder => policyBuilder
                .RequireClaim(Claims.BoardingPassNumber));
        options.AddPolicy(
            "CanAccessLounge",
```

```

        policyBuilder => policyBuilder.AddRequirements(
            new MinimumAgeRequirement(18),
            new AllowedInLoungeRequirement()
        ));
    };
    services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();
    services.AddSingleton<IAuthorizationHandler, FrequentFlyerHandler>();
    services
        .AddSingleton<IAuthorizationHandler, BannedFromLoungeHandler>();
    services
        .AddSingleton<IAuthorizationHandler, IsAirlineEmployeeHandler>();
    // Additional service configuration
}

```

For this app, the handlers don't have any constructor dependencies, so I've registered them as singletons with the container. If your handlers have scoped or transient dependencies (the EF Core DbContext, for example), you might want to register them as scoped instead, as appropriate.

NOTE Services are registered with a lifetime of either transient, scoped, or singleton, as discussed in chapter 10.

You can combine the concepts of policies, requirements, and handlers in many ways to achieve your goals for authorization in your application. The example in this section, although contrived, demonstrates each of the components you need to apply authorization declaratively at the action method or Razor Page level, by creating policies and applying the [Authorize] attribute as appropriate.

As well as applying the [Authorize] attribute explicitly to actions and Razor Pages, you can also configure it globally, so that a policy is applied to every Razor Page or controller in your application. Additionally, for Razor Pages you can apply different authorization policies to different folders. You can read more about applying authorization policies using conventions in Microsoft's "Razor Pages authorization conventions in ASP.NET Core" documentation: <http://mng.bz/nMm2>.

There's one area, however, where the [Authorize] attribute falls short: resource-based authorization. The [Authorize] attribute attaches metadata to an endpoint, so the authorization middleware can authorize the user *before* an endpoint is executed, but what if you need to authorize the action *during* the action method or Razor Page handler?

This is common when you're applying authorization at the document or resource level. If users are only allowed to edit documents they created, then you need to load the document before you can tell whether they're allowed to edit it! This isn't easy with the declarative [Authorize] attribute approach, so you must use an alternative, imperative approach. In the next section, you'll see how to apply this resource-based authorization in a Razor Page handler.

15.5 Controlling access with resource-based authorization

In this section you'll learn about resource-based authorization. This is used when you need to know details about the resource being protected to determine if a user is authorized. You'll learn how to apply authorization policies manually using the `IAuthorizationService`, and how to create resource-based `AuthorizationHandlers`.

Resource-based authorization is a common problem for applications, especially when you have users who can create or edit some sort of document. Consider the recipe application you built in the previous three chapters. This app lets users create, view, and edit recipes.

Up to this point, everyone can create new recipes, and anyone can edit any recipe, even if they haven't logged in. Now you want to add some additional behavior:

- Only authenticated users should be able to create new recipes.
- You can only edit the recipes you created.

You've already seen how to achieve the first of these requirements: decorate the `Create.cshtml` Razor Page with an `[Authorize]` attribute and don't specify a policy, as shown in this listing. This will force the user to authenticate before they can create a new recipe.

Listing 15.12 Adding `AuthorizeAttribute` to the `Create.cshtml` Razor Page

```
[Authorize]
public class CreateModel : PageModel
{
    [BindProperty]
    public CreateRecipeCommand Input { get; set; }

    public void OnGet()
    {
        Input = new CreateRecipeCommand();
    }

    public async Task<IActionResult> OnPost()
    {
        // Method body not shown for brevity
    }
}
```

← Users must be authenticated to execute the `Create.cshtml` Razor Page.

All page handlers are protected. You can only apply `[Authorize]` to the `PageModel`, not handlers.

TIP As with all filters, you can only apply the `[Authorize]` attribute to the Razor Page, not to individual page handlers. The attribute applies to all page handlers in the Razor Page.

Adding the `[Authorize]` attribute fulfills your first requirement, but unfortunately, with the techniques you've seen so far, you have no way to fulfill the second. You could apply a policy that either permits or denies a user the ability to edit *all* recipes, but there's currently no easy way to restrict this so that a user can only edit *their own* recipes.

In order to find out who created the Recipe, you must first load it from the database. Only then can you attempt to authorize the user, taking the specific recipe (resource) into account. The following listing shows a partially implemented page handler for how this might look, where authorization occurs partway through the method, after the Recipe object has been loaded.

Listing 15.13 The Edit.cshtml page must load the Recipe before authorizing the request

```
public IActionResult OnGet(int id)
{
    var recipe = _service.GetRecipe(id);
    var createdById = recipe.CreatedById;
    // Authorize user based on createdById
    if (isAuthorized)
    {
        return View(recipe);
    }
}
```

The id of the recipe to edit is provided by model binding.

You must load the Recipe from the database before you know who created it.

The action method can only continue if the user was authorized.

You must authorize the current user to verify they're allowed to edit this specific Recipe.

You need access to the resource (in this case, the Recipe entity) to perform the authorization, so the declarative `[Authorize]` attribute can't help you. In section 15.5.1 you'll see the approach you need to take to handle these situations and to apply authorization inside the action method or Razor Page.

WARNING Be careful when exposing the integer ID of your entities in the URL, as in listing 15.13. Users will be able to edit every entity by modifying the ID in the URL to access a different entity. Be sure to apply authorization checks, or you could expose a security vulnerability called *insecure direct object reference* (IDOR).²

15.5.1 Manually authorizing requests with `IAuthorizationService`

All of the approaches to authorization so far have been *declarative*. You apply the `[Authorize]` attribute, with or without a policy name, and you let the framework take care of performing the authorization itself.

For this recipe-editing example, you need to use *imperative* authorization, so you can authorize the user after you've loaded the Recipe from the database. Instead of applying a marker saying, "Authorize this method," you need to write some of the authorization code yourself.

DEFINITION *Declarative* and *imperative* are two different styles of programming. Declarative programming describes *what* you're trying to achieve and lets the framework figure out how to achieve it. Imperative programming describes *how* to achieve something by providing each of the steps needed.

² You can read about insecure direct object reference (IDOR) and ways to counteract it on the Open Web Application Security Project (OWASP): <https://owasp.org/www-chapter-ghana/assets/slides/IDOR.pdf>.

ASP.NET Core exposes `IAuthorizationService`, which you can inject into your Razor Pages and controllers for imperative authorization. The following listing shows how you can update the `Edit.cshtml` Razor Page (shown partially in listing 15.13) to use the `IAuthorizationService` and verify whether the action is allowed to continue execution.

Listing 15.14 Using `IAuthorizationService` for resource-based authorization

```
[Authorize]
public class EditModel : PageModel
{
    [BindProperty]
    public Recipe Recipe { get; set; }

    private readonly RecipeService _service;
    private readonly IAuthorizationService _authService;

    public EditModel(
        RecipeService service,
        IAuthorizationService authService)
    {
        _service = service;
        _authService = authService;
    }

    public async Task<IActionResult> OnGet(int id)
    {
        Recipe = _service.GetRecipe(id);
        var authResult = await _authService
            .AuthorizeAsync(User, Recipe, "CanManageRecipe");
        if (!authResult.Succeeded)
        {
            return new ForbidResult();
        }

        return Page();
    }
}
```

Only authenticated users should be allowed to edit recipes.

IAuthorizationService is injected into the class constructor using DI.

Load the Recipe from the database.

Calls IAuthorizationService, providing ClaimsPrincipal, resource, and the policy name

If authorization failed, returns a Forbidden result

If authorization was successful, continues displaying the Razor Page

`IAuthorizationService` exposes an `AuthorizeAsync` method, which requires three things to authorize the request:

- The `ClaimsPrincipal` user object, exposed on the `PageModel` as `User`
- The resource being authorized: `Recipe`
- The policy to evaluate: `"CanManageRecipe"`

The authorization attempt returns an `AuthorizationResult` object, which indicates whether the attempt was successful via the `Succeeded` property. If the attempt wasn't successful, you should return a new `ForbidResult`, which will be converted either into an HTTP 403 Forbidden response or will redirect the user to the "access denied"

page, depending on whether you're building a traditional web app with Razor Pages or a Web API.

NOTE As mentioned in section 15.2.2, which type of response is generated depends on which authentication services are configured. The default Identity configuration, used by Razor Pages, generates redirects. The JWT bearer token authentication typically used with Web APIs generates HTTP 401 and 403 responses instead.

You've configured the imperative authorization in the `Edit.cshtml` Razor Page itself, but you still need to define the "CanManageRecipe" policy that you use to authorize the user. This is the same process as for declarative authorization, so you have to do the following:

- Create a *policy* in `ConfigureServices` by calling `AddAuthorization()`
- Define one or more *requirements* for the policy
- Define one or more *handlers* for each requirement
- Register the handlers in the DI container

With the exception of the handler, these steps are all identical to the declarative authorization approach with the `[Authorize]` attribute, so I'll only run through them briefly here.

First, you can create a simple `IAuthorizationRequirement`. As with many requirements, this contains no data and simply implements the marker interface.

```
public class IsRecipeOwnerRequirement : IAuthorizationRequirement { }
```

Defining the policy in `ConfigureServices` is similarly simple, as you have only this single requirement. Note that there's nothing resource-specific in any of this code so far:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthorization(options => {
        options.AddPolicy("CanManageRecipe", policyBuilder =>
            policyBuilder.AddRequirements(new IsRecipeOwnerRequirement()));
    });
}
```

You're halfway there; all you need to do now is create an authorization handler for `IsRecipeOwnerRequirement` and register it with the DI container.

15.5.2 Creating a resource-based AuthorizationHandler

Resource-based authorization handlers are essentially the same as the authorization handler implementations you saw in section 15.4.2. The only difference is that the handler also has access to the resource being authorized.

To create a resource-based handler, you should derive from the `AuthorizationHandler<TRequirement, TResource>` base class, where `TRequirement` is the type of

requirement to handle, and `TResource` is the type of resource that you provide when calling `IAuthorizationService`. Compare this to the `AuthorizationHandler<T>` class you implemented previously, where you only specified the requirement.

This listing shows the handler implementation for your recipe application. You can see that you've specified the requirement as `IsRecipeOwnerRequirement` and the resource as `Recipe`, and you have implemented the `HandleRequirementAsync` method.

Listing 15.15 `IsRecipeOwnerHandler` for resource-based authorization

```
public class IsRecipeOwnerHandler :
    AuthorizationHandler<IsRecipeOwnerRequirement, Recipe>
{
    private readonly UserManager<ApplicationUser> _userManager;
    public IsRecipeOwnerHandler(
        UserManager<ApplicationUser> userManager)
    {
        _userManager = userManager;
    }
    protected override async Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        IsRecipeOwnerRequirement requirement,
        Recipe resource)
    {
        var appUser = await _userManager.GetUserAsync(context.User);
        if (appUser == null)
        {
            return;
        }
        if (resource.CreatedById == appUser.Id)
        {
            context.Succeed(requirement);
        }
    }
}
```

Implements the necessary base class, specifying the requirement and resource type

Injects an instance of the `UserManager<T>` class using DI

As well as the context and requirement, you're also provided the resource instance.

If you aren't authenticated, `appUser` will be null.

Checks whether the current user created the Recipe by checking the `CreatedById` property

If the user created the document, Succeed the requirement; otherwise, do nothing.

This handler is slightly more complicated than the examples you've seen previously, primarily because you're using an additional service, `UserManager<>`, to load the `ApplicationUser` entity based on `ClaimsPrincipal` from the request.

NOTE In practice, the `ClaimsPrincipal` will likely already have the `Id` added as a claim, making the extra step unnecessary in this case. This example shows the general pattern if you need to use dependency-injected services.

The other significant difference is that the `HandleRequirementAsync` method has provided the `Recipe` resource as a method argument. This is the same object that you provided when calling `AuthorizeAsync` on `IAuthorizationService`. You can use this resource to verify whether the current user created it. If so, you `Succeed()` the requirement; otherwise you do nothing.

The final task is to add `IsRecipeOwnerHandler` to the DI container. Your handler uses an additional dependency, `userManager<>`, which uses EF Core, so you should register the handler as a scoped service:

```
services.AddScoped<IAuthorizationHandler, IsRecipeOwnerHandler>();
```

TIP If you're wondering how to know whether you register a handler as scoped or a singleton, think back to chapter 10. Essentially, if you have scoped dependencies, you must register the handler as scoped; otherwise singleton is fine.

With everything hooked up, you can take the application for a spin. If you try to edit a recipe you didn't create by clicking the Edit button on the recipe, you'll either be redirected to the login page (if you hadn't yet authenticated) or you'll be presented with an "access denied" page, as shown in figure 15.7.

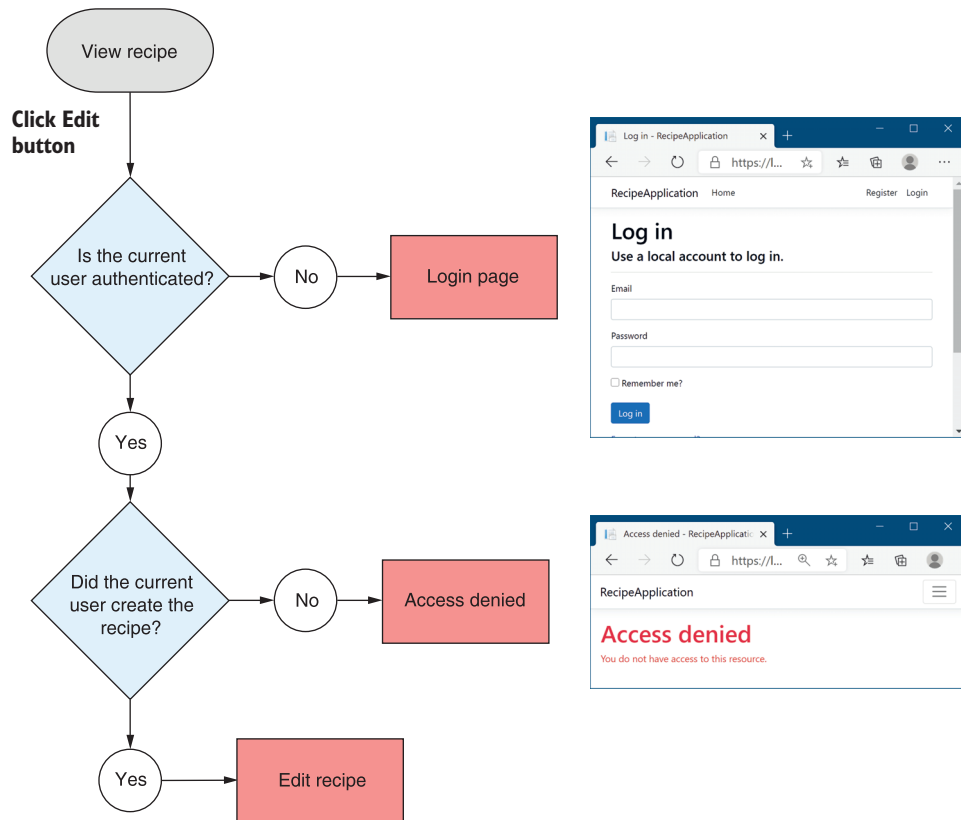


Figure 15.7 If you're logged in but not authorized to edit a recipe, you'll be redirected to an "access denied" page. If you're not logged in, you'll be redirected to the login page.

By using resource-based authorization, you're able to enact more fine-grained authorization requirements that you can apply at the level of an individual document or resource. Instead of only being able to authorize that a user can edit *any* recipe, you can authorize whether a user can edit *this* recipe.

Resource-based authorization versus business-logic checks

The value proposition of using the ASP.NET Core framework's resource-based authorization approach isn't always clear when compared to using simple, manual, business-logic based checks (as in listing 15.13). Using `IAuthorizationService` and the authorization infrastructure adds an explicit dependency on the ASP.NET Core framework that you may not want to use if you're performing authorization checks in your domain model services.

This is a valid concern without an easy answer. I tend to favor simple business-logic checks inside the domain, without relying on the framework's authorization infrastructure, to make my domain easier to test and framework-independent. But doing so loses some of the benefits of such a framework:

- The `IAuthorizationService` uses declarative policies, even though you are calling the authorization framework imperatively.
- You can decouple the need to authorize an action from the actual requirements.
- You can easily rely on peripheral services and properties of the request, which may be harder (or undesirable) with business logic checks.

You *can* achieve these benefits in business-logic checks, but that typically requires creating a lot of infrastructure too, so you lose a lot of the benefits of keeping things simple. Which approach is best will depend on the specifics of your application design, and there may well be cases for using both.

All the authorization techniques you've seen so far have focused on server-side checks. Both the `[Authorize]` attribute and resource-based authorization approaches focus on stopping users from executing a protected action on the server. This is important from a security point of view, but there's another aspect you should consider too: the user experience when they don't have permission.

You've protected the code executing on the server, but arguably the Edit button should never have been visible to the user if they weren't going to be allowed to edit the recipe! In the next section we'll look at how you can conditionally hide the Edit button by using resource-based authorization in your view models.

15.6 Hiding elements in Razor templates from unauthorized users

All the authorization code you've seen so far has revolved around protecting action methods or Razor Pages on the server side, rather than modifying the UI for users. This is important and should be the starting point whenever you add authorization to an app.

WARNING Malicious users can easily circumvent your UI, so it's important to always authorize your actions and Razor Pages on the server, never on the client alone.

From a user-experience point of view, however, it's not friendly to have buttons or links that look like they're available, but which present you with an “access denied” page when they're clicked. A better experience would be for the links to be disabled, or not visible at all.

You can achieve this in several ways in your own Razor templates. In this section, I'm going to show you how to add an additional property to the `PageModel`, called `CanEditRecipe`, which the Razor view template will use to change the rendered HTML.

TIP An alternative approach would be to inject `IAuthorizationService` directly into the view template using the `@inject` directive, as you saw in chapter 10, but you should prefer to keep logic like this in the page handler.

When you're finished, the rendered HTML will look unchanged for recipes you created, but the Edit button will be hidden when viewing a recipe someone else created, as shown in figure 15.8.

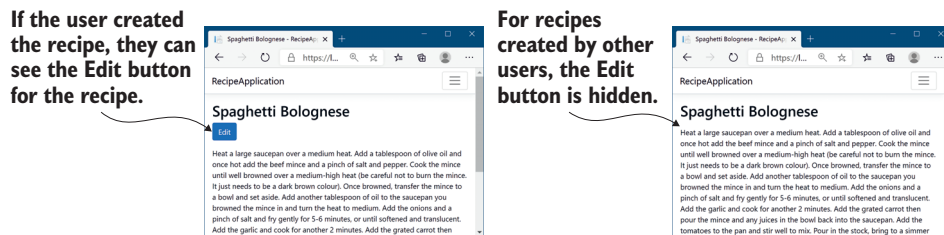


Figure 15.8 Although the HTML will appear unchanged for recipes you created, the Edit button is hidden when you view recipes created by a different user.

The following listing shows the `PageModel` for the `View.cshtml` Razor Page, which is used to render the recipe page shown in figure 15.8. As you've already seen for resource-based authorization, you can use the `IAuthorizationService` to determine whether the current user has permission to edit the `Recipe` by calling `AuthorizeAsync`. You can then set this value as an additional property on the `PageModel`, called `CanEditRecipe`.

Listing 15.16 Setting the `CanEditRecipe` property in the `View.cshtml` Razor Page

```
public class ViewModel : PageModel
{
    public Recipe Recipe { get; set; }
    public bool CanEditRecipe { get; set; }
```

The `CanEditRecipe` property will be used to control whether the Edit button is rendered.

```

private readonly RecipeService _service;
private readonly IAuthorizationService _authService;
public ViewModel(
    RecipeService service,
    IAuthorizationService authService)
{
    _service = service;
    _authService = authService;
}

public async Task<IActionResult> OnGetAsync(int id)
{
    Recipe = _service.GetRecipe(id);
    var isAuthorised = await _authService
        .AuthorizeAsync(User, recipe, "CanManageRecipe");
    CanEditRecipe = isAuthorised.Succeeded;
    return Page();
}

```

Loads the Recipe resource for use with IAuthorizationService

Verifies whether the user is authorized to edit the Recipe

Sets the CanEditRecipe property on the PageModel as appropriate

Instead of blocking execution of the Razor Page (as you did previously in the Edit.cshtml page handler), use the result of the call to `AuthorizeAsync` to set the `CanEditRecipe` value on the `PageModel`. You can then make a simple change to the `View.chhtml` Razor template: add an `if` clause around the rendering of the Edit link.

```

@if (Model.CanEditRecipe)
{
    <a asp-page="Edit" asp-route-id="@Model.Id"
        class="btn btn-primary">Edit</a>
}

```

This ensures that only users who will be able to execute the `Edit.cshtml` Razor Page can see the link to that page.

WARNING The `if` clause means the Edit link will not be displayed unless the user created the recipe, but *a malicious user can still circumvent your UI*. It's important to keep the server-side authorization check in your `Edit.cshtml` page handler to protect against these circumvention attempts.

With that final change, you've finished adding authorization to the recipe application. Anonymous users can browse the recipes created by others, but they must log in to create new recipes. Additionally, authenticated users can only edit the recipes that they created, and they won't see an Edit link for other people's recipes.

Authorization is a key aspect of most apps, so it's important to bear it in mind from an early point. Although it's possible to add authorization later, as you did with the recipe app, it's normally preferable to consider authorization sooner rather than later in the app's development.

In the next chapter we're going to be looking at your ASP.NET Core application from a different point of view. Instead of focusing on the code and logic behind your app, we're going to look at how you prepare an app for production. You'll see how to

specify the URLs your application uses and how to publish an app so that it can be hosted in IIS. Finally, you'll learn about the bundling and minification of client-side assets, why you should care, and how to use `BundlerMinifier` in ASP.NET Core.

Summary

- Authentication is the process of determining who a user is. It's distinct from authorization, the process of determining what a user can do. Authentication typically occurs before authorization.
- You can use the authorization services in any part of your application, but it's typically applied using the `AuthorizationMiddleware` by calling `UseAuthorization()`. This should be placed after the calls to `UseRouting()` and `UseAuthentication()`, and before the call to `UseEndpoints()` for correct operation.
- You can protect Razor Pages and MVC actions by applying the `[Authorize]` attribute. The routing middleware records the presence of the attribute as metadata with the selected endpoint. The authorization middleware uses this metadata to determine how to authorize the request.
- The simplest form of authorization requires that a user be authenticated before executing an action. You can achieve this by applying the `[Authorize]` attribute to a Razor Page, action, controller, or globally. You can also apply attributes conventionally to a subset of Razor Pages.
- Claims-based authorization uses the current user's claims to determine whether they're authorized to execute an action. You define the claims needed to execute an action in a *policy*.
- Policies have a name and are configured in `Startup.cs` as part of the call to `AddAuthorization()` in `ConfigureServices`. You define the policy using `AddPolicy()`, passing in a name and a lambda that defines the claims needed.
- You can apply a policy to an action or Razor Page by specifying the policy in the `authorize` attribute; for example, `[Authorize("CanAccessLounge")]`. This policy will be used by the `AuthorizationMiddleware` to determine if the user is allowed to execute the selected endpoint.
- In a Razor Pages app, if an unauthenticated user attempts to execute a protected action, they'll be redirected to the login page for your app. If they're already authenticated but don't have the required claims, they'll be shown an "access denied" page instead.
- For complex authorization policies, you can build a custom policy. A custom policy consists of one or more requirements, and a requirement can have one or more handlers. You can combine requirements and handlers to create policies of arbitrary complexity.
- For a policy to be authorized, every requirement must be satisfied. For a requirement to be satisfied, one or more of the associated handlers must indicate success, and none must indicate explicit failure.

- `AuthorizationHandler<T>` contains the logic that determines whether a requirement is satisfied. For example, if a requirement requires that users be over 18, the handler could look for a `DateOfBirth` claim and calculate the user's age.
- Handlers can mark a requirement as satisfied by calling `context.Succeed(requirement)`. If a handler can't satisfy the requirement, then it shouldn't call anything on the context, as a different handler could call `Succeed()` and satisfy the requirement.
- If a handler calls `context.Fail()`, the requirement will fail, even if a different handler marked it as a success using `Succeed()`. Only use this method if you want to override any calls to `Succeed()` from other handlers, to ensure the authorization policy will fail authorization.
- Resource-based authorization uses details of the resource being protected to determine whether the current user is authorized. For example, if a user is only allowed to edit their own documents, you need to know the author of the document before you can determine whether they're authorized.
- Resource-based authorization uses the same policy, requirements, and handler system as before. Instead of applying authorization with the `[Authorize]` attribute, you must manually call `IAuthorizationService` and provide the resource you're protecting.
- You can modify the user interface to account for user authorization by adding additional properties to your `PageModel`. If a user isn't authorized to execute an action, you can remove or disable the link to that action method in the UI. You should always authorize on the server, even if you've removed links from the UI.